True Lies: Lazy Contracts for Lazy Languages

Faithfulness is Better than Laziness

Markus Degen Peter Thiemann Stefan Wehr
{degen,thiemann,wehr}@informatik.uni-freiburg.de

Abstract: Contracts are a proven tool in software development. They provide specifications for operations that may be statically verified or dynamically validated by contract monitoring.

Contract monitoring for lazy programming languages does not have a generally accepted basis. This paper discusses three approaches, eager, semi-eager, and lazy monitoring, all of which are flawed. The first two may change program behavior, while the last two may lead to silent contract violations.

1 Introduction

Design by contract [Mey97] is a methodology for constructing correct software. It attaches a contract to each operation. Such a contract consists of a precondition and a postcondition: All input must fulfill the precondition and then the output is also obliged to meet the postcondition. Thus, a contract provides a (partial) specification of an operation and an implementation of the operation must fulfill that contract.

While contracts can be verified statically, in practice, they are mostly checked dynamically using contract monitoring (cf. Eiffel [Mey92b, Mey92a], Scheme [PLT05], Java [AK03, Kra98], or Haskell [HJL06]): Each operation checks the precondition before performing its computation and checks the postcondition before returning to its caller. On violation of the precondition the operation raises an exception blaming its caller. Conversely, on violation of the postcondition the operation blames itself.

The semantics of contract monitoring is intricate, and its correct and complete implementation is non-trivial [FF01] (see Section 6 for further discussion on related work). From a practical point of view, contract monitoring should guarantee meaning reflection (MR) and meaning preservation (MP), report contract violations faithfully (F), and behave idempotently (IP):

MR If running a program $\mathcal{P}$ with contract monitoring terminates and delivers a result v, then running $\mathcal{P}$ without contract monitoring also terminates and delivers the same result v. The MR property ensures that developers may enable contract monitoring for a test version of their software and safely disable contract monitoring for the release version, without running the risk that the test and the release version behave differently.

MP If running a program $\mathcal{P}$ without contract monitoring terminates and delivers a result v, then running $\mathcal{P}$ with contract monitoring also terminates and delivers the same result v or it signals a contract violation. The MP property ensures that contracts may be added to an existing program incrementally, without running the risk of changing its behavior.

F If running a program terminates and delivers a result v after progressing through intermediate states $\mathcal{E}_i[!C_i\, e_i]$ (a contract C_i applied to some expression e_i in some context $\mathcal{E}_i$), then the predicates involved in $!C_i\, e_i$ are all true. The F property ensures that programmer can rely on the truth of the predicates contained in a contract.

IP If enforcing contract C on expression e yields result e', then enforcing C twice on e also yields e'. The IP property ensures a meaningful notion for contract composition.

In the context of call-by-value evaluation, there is only one useful and sensible mode of contract monitoring, which we call *eager monitoring*. This mode corresponds to the implementation of monitoring in Eiffel, Java, and Scheme as outlined in the second paragraph. The four properties MR, MP, F, and IP hold for eager contract monitoring under call-by-value evaluation, provided contract predicates are side-effect free.

For lazy languages, it turns out that there are in principle three conceivable modes of monitoring:

- *Eager monitoring* evaluates all predicates of non-function contracts as soon as the contract is enforced. For function contracts, eager monitoring enforces the precondition of a function argument before evaluating the body. Similarly, it enforces the postcondition before returning the result.
- *Semi-eager monitoring* implements contract enforcement for non-function contracts by evaluating only those predicates of a contract that correspond to expressions demanded by the program. For function contracts, semi-eager monitoring does not enforce the precondition unless the function demands the argument (*e.g.*, unless it is strict). Similarly, enforcement of the postcondition is driven by evaluation of the function's result. This mode of monitoring is roughly what Hinze et al. propose [HJL06].
- *Lazy monitoring* coincides with semi-eager monitoring with respect to function contracts. For non-function contracts, however, lazy monitoring evaluates a predicate of a contract only if all values on which the predicate depends are already evaluated. That is, lazy monitoring only proceeds as far as the user code itself can observe. Violations that the user code cannot observe, yet, are considered to be invisible. Chitil and Huch's pattern logic [CH07b] approximates lazy monitoring for non-function contracts.

Contributions. We have identified a taxonomy of three different approaches to contract monitoring in lazy languages, thus arriving at a clearly defined design space for this prob-

$$
\begin{array}{c}
e ::= c \mid x \mid () \mid \mathtt{rec}\, f(x) = e \mid e\, e \mid (e, e) \mid \mathtt{case}\, e\, \mathtt{of}\, (x_1, x_2) : e \\
\quad \mid \mathtt{in}_i\, e \mid \mathtt{case}\, e\, \mathtt{of}\, \mathtt{in}_1\, x : e; \mathtt{in}_2\, x : e \mid !C\, e \\
C ::= ?\xi.P \mid H_{\rightarrow}\, C\, C \mid H_{\times}\, C\, C \mid H_{+}\, C\, C \mid H_{\&}\, C\, C \\
P ::= e \mid \mathtt{CASE}\, \xi\, \mathtt{OF}\, (\xi, \xi) : P \mid \mathtt{CASE}\, \xi\, \mathtt{OF}\, \mathtt{IN}_1\, \xi : P; \mathtt{IN}_2\, \xi : P \\
c \in \mathcal{C}\mathrm{onst}, \quad x, f \in \mathcal{V}\mathrm{ar}, \quad \xi \in \mathcal{BV}\mathrm{ar} \subseteq \mathcal{V}\mathrm{ar}
\end{array}
$$

Figure 1: A language with contracts

lem. We provide implementations for the novel notion of eager monitoring as well as for lazy monitoring, using a novel implementation strategy.

We measure the effectiveness and usefulness of the approaches in our taxonomy along five dimensions: meaning reflection, meaning preservation, faithfulness, idempotence, and implementability. The central insight is that contract monitoring in a lazy language cannot be faithful and meaning preserving at the same time.

Overview. Section 2 introduces a lambda calculus notation enriched with contract primitives. This syntax is the basis for our subsequent discussion of the taxonomy in Section 3. This discussion is structured in Subsections 3.1 (eager monitoring), 3.2 (Semi-eager monitoring), and 3.3 (lazy monitoring). Each of these sections illustrates problems with one or several of the dimensions through examples. Section 4 considers ways of implementing the three approaches. Section 5 summarizes the problems of the approaches, puts them in perspective, and recommends that contracts should be faithful. The paper concludes with a review of related work (Section 6) and a conclusion (Section 7).

2 A Language with Contracts

To provide a common syntactic basis for our investigation, we define a small lambda calculus-based language, $\lambda^{?!}$, with contracts and the standard provisions for sums, products, and recursive functions. Figure 1 defines its syntax. Expressions e are mostly standard. *Contract enforcement*, $!C\, e$, asserts that the result of e fulfills contract C. We use ξ to range over *box variables*. For eager and semi-eager monitoring, box variables are equivalent to regular variables. For lazy monitoring, however, box variables have a special status (see Sections 3.3 and 4.3).

There are five forms of contracts. A *predicate contract* $?\xi.P$ binds ξ to a value v and enforces the pattern predicate P on v. The value of the contract enforcement is v if P evaluates true. Otherwise, it aborts execution and signals a contract violation. A *function contract* $H_{\rightarrow}\, C_1\, C_2$ confines a function value to have precondition C_1 and postcondition C_2. A *pair contract* $H_{\times}\, C_1\, C_2$ (*sum contract* $H_{+}\, C_1\, C_2$) restricts the components of a pair (sum) according to C_1 and C_2. Finally, $H_{\&}\, C_1\, C_2$ denotes the *conjunction* of C_1 and C_2.

A pattern predicate P is either one of the special elimination constructs that decompose a

$$
\begin{array}{lcl}
\mathsf{ordered} & \equiv & ?\xi.\mathtt{CASE}\,\xi\,\mathtt{OF}\,(\xi_1,\xi_2):\ \xi_1 < \xi_2 \\
\mathsf{arg\text{-}is\text{-}zero} & \equiv & H_{\to}\,(?\xi.\xi = 0)\,(?\xi.\mathsf{true}) \\
\mathsf{first\text{-}zero\text{-}and\text{-}ordered} & \equiv & H_{\&}\,(H_{\times}\,(?\xi.\xi = 0)\,(?\xi.\mathsf{true}))\,\mathsf{ordered}
\end{array}
$$

Figure 2: Example contracts.

pair or a sum, or a boolean expression. The boolean expression must not contain further (standard) elimination constructs. For eager and semi-eager monitoring, the special elimination constructs behave like the standard elimination constructs. For lazy monitoring, they behave differently (*e.g.*, lazily).

The dynamic semantics of the expression language (except contract enforcement $!C\,e$) is based on the principle of lazy evaluation as prescribed, *e.g.*, by Launchbury's operational semantics [Lau93]. We explain the semantics of the various strategies for evaluating $!C\,e$ in their dedicated sections.

Notation. We write $\lambda x.e$ instead of $\mathtt{rec}\,f(x) = e$ if f is not free in e and let $\equiv$ indicate syntactic equality. We set $\mathsf{true} \equiv \mathtt{in}_1\,()$ and $\mathsf{false} \equiv \mathtt{in}_2\,()$ to define the conditional: $\mathsf{if}\,e\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2 \equiv \mathtt{case}\,e\,\mathtt{of}\,\mathtt{in}_1\,x_1 : e_1; \mathtt{in}_2\,x_2 : e_2$ where x_1, x_2 are fresh.

Examples. Figure 2 defines a few example contracts that serve as running examples to illustrate the problems occurring with eager, semi-eager, and lazy contract monitoring.

1. The contract ordered asserts that the first component of a pair is smaller than the second.
2. The contract arg-is-zero asserts that the argument of the function is zero, leaving the result unrestricted
3. The contract first-zero-and-ordered ensures that the first component of a pair is zero and that the second component is greater than the first.

3 Three Styles of Contract Monitoring

This section considers each of the identified styles of contract monitoring in turn. Each subsection introduces one style, explains how it works, and evaluates the four dimensions meaning reflection, meaning preservation, faithfulness, and idempotence. Section 4 considers implementation matters.

3.1 Eager Contract Monitoring

Eager contract monitoring evaluates all predicates of non-function contracts as soon as the contract is enforced. For function contracts, eager monitoring enforces the precondition of a function argument before evaluating the body. Similarly, it enforces the postcondition before returning the result.

Fully evaluating contract predicates ensures faithfulness and idempotence. Moreover, the MR property holds because our language $\lambda^{?!}$ has no side-effects except non-termination. However, eager contract monitoring may violate the MP property. Here is an example that reveals the problem:[1]

$$(!\mathsf{arg\text{-}is\text{-}zero}\ (\lambda x.1))\ \bot \tag{1}$$

This expression diverges with contract enforcement because the predicate of the contract arg-is-zero forces the argument $\bot$. With contract monitoring disabled, this expression is equivalent to $(\lambda x.1)\ \bot$, which evaluates to 1. Hence, MP is violated.

3.2 Semi-eager Contract Monitoring

Semi-eager contract monitoring implements contract enforcement for non-function contracts by evaluating only those predicates of a contract that correspond to expressions demanded by the program. For function contracts, semi-eager monitoring does not enforce the precondition unless the function demands the argument (*e.g.*, unless it is strict). Similarly, enforcement of the postcondition is driven by evaluation of the function's result.

With the same argument as for eager monitoring, the MR property also holds for semi-eager contracts monitoring. Showing that MP does not hold requires a slightly more elaborate program than in Example (1):

$$\texttt{case}\ (!\mathsf{ordered}\ (5, \bot))\ \texttt{of}\ (x_1, x_2) : x_1 \tag{2}$$

This expression diverges because the `case` expression forces ordered's predicate to be evaluated, which in turn forces evaluation of the second pair component $\bot$. Removing the contract yields $\texttt{case}\ (5, \bot)\ \texttt{of}\ (x_1, x_2) : x_1$ which evaluates to 5.

Semi-eager contract monitoring also violates the faithfulness property F. As Meyer advocates [Mey92a], it is considered bad coding style to defensively check the parameter at the beginning of the function. Consider a constant function that relies on the arg-is-zero contract to restrict its domain.

$$\mathsf{my\text{-}const} \equiv !\mathsf{arg\text{-}is\text{-}zero}\ (\lambda x.42) \tag{3}$$

The programmer decides to rely on contracts and implements the function without checking the domain again. Still, executing $(\mathsf{my\text{-}const}\ 5)$ yields 42. This undesired behavior is a direct consequence of semi-eager monitoring, which only checks the precondition of

[1] $\bot$ stands for a non-terminating expression.

a function if the function demands its argument. As this constant function never evaluates its argument, semi-eager contract monitoring never checks the precondition arg-is-zero. This behavior may lead to subtle bugs in programs whenever the programmer trusts the contract system to check the provided preconditions.

As pointed out by Hinze et al. [HJL06], contract conjunction is—in general—not idempotent under semi-eager contract monitoring. As an example, consider the expression

$$\texttt{case}\ !\mathsf{first\text{-}zero\text{-}and\text{-}ordered}\ (1,2)\ \texttt{of}\ (x_1,x_2) : \mathsf{false}$$

Hinze et al. implement contract conjunction through contract composition; that is, for checking $H_{\&}\, C_1\, C_2$ they first check C_2 and then C_1. Thus, the expression above evaluates to false because the context never demands the erroneous pair component 1. However, if a program enforces first-zero-and-ordered twice, then the second enforcement of ordered demands this component of the pair. Hence, the expression

$$\texttt{case}\ !\mathsf{first\text{-}zero\text{-}and\text{-}ordered}\ (!\mathsf{first\text{-}zero\text{-}and\text{-}ordered}\ (1,2))\ \texttt{of}\ (x_1,x_2) : \mathsf{false} \quad (4)$$

causes a contract violation.

3.3 Lazy Contract Monitoring

Lazy contract monitoring coincides with semi-eager monitoring with respect to function contracts. For non-function contracts, however, lazy monitoring evaluates a predicate of a contract only if all values on which the predicate depends are already evaluated. That is, lazy monitoring only proceeds as far as the user code itself can observe. Violations that the user code cannot observe, yet, are considered to be invisible.

To allow for lazy contract monitoring, contracts do no longer access expressions under observation directly but through special reference cells called *boxes*. A box is empty as long as its expression has not been reduced to a value, otherwise it carries the value. If contract enforcement finds a box empty, it delays evaluation of the contract until this box is filled. This magic is hidden by accessing boxes only through special box variables, ranged over by ξ. The patterns

$$\texttt{CASE}\,\xi\,\texttt{OF}\ (\xi_1,\xi_2) : \ldots \qquad \text{and} \qquad \texttt{CASE}\,\xi\,\texttt{OF}\ \texttt{IN}_1\,\xi_1 : \ldots ; \texttt{IN}_2\,\xi_2 : \ldots$$

operate on boxes. The evaluation of either pattern is blocked until the box ξ becomes full. As soon as ξ is full, ξ_1 and ξ_2 are bound to boxes for the components of the value in ξ.

Like the two other forms of contract monitoring, lazy monitoring guarantees the MR property. Lazy monitoring also ensures the MP and the IP properties because it never forces any extra evaluation beyond that required by the program's execution. Consequently, Examples (1), (2), and (4) terminate and yield 1, 5, and false, respectively.

For the same reason as with semi-eager contract monitoring, Example (3) still yields 42 for all inputs. Thus, lazy contract monitoring violates the faithfulness property F.

```
data Contract :: * -> * where
  Prop ::  (a -> Bool) -> Contract a
  Pair ::  Contract a -> Contract b -> Contract (a, b)
  Fun  ::  Contract a -> Contract b -> Contract (a -> b)

assert ::  Contract a -> a -> a
assert c = \ a -> let (b, a') = assert' c a in a'

assert' :: Contract a -> a -> (Bool, a)
assert' (Prop p)     =
  \ a -> let b = p a in (b, if b then a else error "assert failed")
assert' (Pair c1 c2) =
  \ (x1, x2) -> let (b1', x1') = assert' c1 x1
                    (b2', x2') = assert' c2 x2
                    r = if not b1' then x1' `seq` error "never used" else
                        if not b2' then x2' `seq` error "never used" else
                        (x1', x2')
                in  (b1' && b2', r)
assert' (Fun c1 c2)  =
   \ f -> (True, \ x -> let (b', x') = assert' c1 x in
                        if b' then
                           snd (assert' c2 (f x'))
                        else
                           x' `seq` error "never used")
```

Figure 3: Haskell implementation of eager monitoring.

4 Implementation

This considers the implementation of the discussed forms of contract monitoring. Two of them, eager and semi-eager monitoring, are readily implementable as user-level libraries, whereas lazy monitoring runs into some non-trivial implementation problems. For concreteness, we display snippets of Haskell code where appropriate. The implementations presented here can be found on our webpage[2], including the previously used examples.

4.1 Eager Monitoring

Let us briefly recap the philosophy of eager monitoring: Asserting a contract for an expression should evaluate the expression as far as there are non-trivial contracts to check. The function contract imposes a (precondition) contract on the argument that must be asserted before entering the function's body. Likewise, the result of the function should not be returned before the postcondition contract has been checked on it.

Figure 3 shows an implementation of eager monitoring in Haskell. It is derived from

[2] `http://proglang.informatik.uni-freiburg.de/projects/contracts/true-lies`

Hinze et al's implementation of semi-eager monitoring [HJL06], which relies on GADTs to guarantee type correct handling of the contracts and to enable type indexed programming. As in their implementation, `Contract a` is the type of contracts for values of type `a`. The constructor `Prop (\ x -> e)` corresponds to our predicate contract $?x.e$, `(Pair c1 c2)` corresponds to $H_\times\, C_1\, C_2$, and `(Fun c1 c2)` to $H_\rightarrow\, C_1\, C_2$.

The code is surprisingly tricky to get right. It must neither be too lazy nor too strict. The predicate contract is straightforward to handle. It just checks the predicate and returns the argument if the predicate is `True`.

The trouble starts with the pair contract `(Pair c1 c2)`. The definition of eager monitoring implies that a pair `(x1, x2)` must not be accepted by a pair contract if a subcontract fails on a component of the pair. Hence, asserting a pair contract requires first asserting its component contracts. However, it is not sufficient to string these computations together using Haskell's `seq` operator[3] as in

```
let x1' = assert c1 x1
    x2' = assert c2 x2
in  x1' `seq` x2' `seq` (x1', x2')
```

This code would be too strict as evidenced by the contract

```
Pair (Pred (const True)) (Pred (const True))
```

This contract should be assertable to any pair including $(\bot, \bot)$, but the pure `seq` implementation rejects that pair. The actual implementation in Figure 3 avoids this overstrictness by introducing `assert'` which returns the boolean result of the predicate paired along with the value. This separation makes it possible to have a non-strict predicate in a component avoid the evaluation of the component. A component is only evaluated if its predicate is `False`, in which case the evaluation yields the error message associated with the first failing predicate. The `error "never used"` avoids the type conflict between the components `x1'`, `x2'`, and the pair`(x1', x2')`.

A similar problem arises with the function contract. The eager interpretation of monitoring requires evaluation of the precondition, but not necessarily evaluation of the argument (if the precondition is non-strict). The solution of the problem works similarly to the one for pairs. Checking of the postcondition is left implicit because the result of a function call is always evaluated (otherwise the function would not have been invoked in the first place).

4.2 Semi-Eager Monitoring

Figure 4 shows the essence of semi-eager monitoring as extracted from Hinze et al's paper [HJL06]. It is essentially a transcription of Findler and Felleisen's higher-order contracts from Scheme to Haskell [FF02]. The assertion of a property `Prop p` works as before. A

[3] `seq :: a -> b -> b` forces the evaluation of its first argument and then returns its second argument.

```
assert ::  Contract a -> a -> a
assert (Prop p)     =  \ a -> if p a then a else error "assertion failed"
assert (Pair c1 c2) =  \ (x1, x2) -> (assert c1 x1, assert c2 x2)
assert (Fun c1 c2)  =  \ f -> assert c2 . f . assert c1
```

Figure 4: Haskell implementation of semi-eager monitoring.

```
data Contract :: * -> * where
  Prop     :: (Box a -> P a) -> Contract a
  Function :: Contract a -> (Box a -> Contract b) -> Contract (a -> b)

data P :: * -> * where
  Pred     :: CM Bool -> P a
  CasePair :: Box (a,b) -> ((Box a, Box b) -> P c) -> P (a,b)

data Box a   -- abstract
data CM a    -- abstract, instance of Monad

demand :: Box a -> CM a
assert :: Contract a -> a -> a
```

Figure 5: Interface of the Haskell implementation of lazy contract monitoring.

pair contract now returns a pair whose components may contain nested (hidden) contract violations. Such contract violations may go undetected as already discussed. Similarly, a function contract for a non-strict function never forces the assertion of its precondition (or parts of it).

The problems with semi-eager monitoring are not solved by making the pair contract strict (*e.g.*, forcing it to evaluate its components) or by making the function strict. Both measures just change the semantics of the pair (function), but do not lead to more contracts being checked.

4.3 Lazy Monitoring

The idea of lazy monitoring is to leave the semantics of a program unchanged, but throw an exception as soon as the program can observe a contract violation. That is, the program is evaluated lazily and the `assert` operation attaches contract predicates as observers to its argument expressions. Then the contract predicates are evaluated in a dataflow manner, based on the availability of their inputs.

We have developed a Haskell implementation of lazy monitoring (see Figure 5) again loosely based on the interface of Hinze et al [HJL06].[4] The intention of the type `Contract a` is still the same, but the details are quite different. To enable the observability of evaluation, the implementation introduces a type `Box a`. A box is an observer of the evaluation

[4] We simplified the interface for the sake of presentation. The real implementation supports blame assignment and additional contract forms.

of an expression of type `a`. The box is empty as long as the expression is unevaluated; it becomes full upon evaluation. Filling of such a box is done by wrapping an observer around the expression under contract, similar as in the HOOD debugger [Gil00]. The observer is notified of the evaluation and fills the box with the value. The `demand` operation reads the box. If the value of the box is available, `demand` returns this value in the `CM` monad and raises an exception otherwise. The contract checker then catches this exception and delays evaluation of the predicate until the value is available.

With this setup, a predicate contract `Prop` is no longer constructed from a function of type `a -> Bool` but from a function of type `Box a -> P a`. A value of type `P a` is a pattern that scrutinizes values of type `a`. The `Pred` constructor takes a value of the monadic type `CM Bool` because predicates must use the function `demand` to read box values. The `demand` function is the only operation of the `CM` monad. A `Function` contract consists of a precondition contract and a function that takes a box with the function argument and delivers the postcondition contract.

A pattern is either a predicate computed from values demanded from the open boxes or it is a `CasePair`. The latter specifies lazy pattern matching on a box variable. It creates two new (empty) boxes for the components of the pattern and leaves those for the rest of the contract to process.

We refrain from going into more detail of the implementation but refer the reader to the webpage[5] that contains the full code.

There is one confession to make about our implementation of lazy contract monitoring. It cannot be implemented at the user level in an accurate and satisfactory manner. Here is why.

Ideally, lazy contract monitoring should evaluate a predicate if the expression it refers to turns into a value. This requirement is not entirely unrealistic because this information is known to a typical implementation [Pey92]. However, this information is not readily accessible to a user-level library. Consider the contract $H_\times\ (?x.x > 0)\ (?y.y > 0)$ taken as precondition for the function $\lambda z.\texttt{case}\ z\ \texttt{of}\ (z_1, z_2) : z_1$. With our implementation, this function (with the precondition) can be applied to $(1, 2 - 2)$ without a contract violation, even though the context of the function application may later evaluate $2 - 2$ to 0, so that the contract violation becomes observable to the program.

Hence, we conclude that lazy contract monitoring can only be implemented satisfactorily as an extension to the run-time system. We leave it to the reader to decide if that is a worthwhile project.

5 Discussion

Table 1 summarizes the characteristics of the three contract monitoring strategies for lazy languages. Unfortunately, no strategy fulfills all our requirements:

[5] `http://proglang.informatik.uni-freiburg.de/projects/contracts/true-lies`

	MR	MP	F	IP	Implementability
Eager	✓	↯ see (1)	✓	✓	✓
Semi-eager	✓	↯ see (2)	↯ see (3)	↯ see (4)	✓
Lazy	✓	✓	↯ see (3)	✓	(✓)

Table 1: Characteristics of different contract monitoring strategies

- Eager contract monitoring may change the meaning of a program.
- Semi-eager contract monitoring may also change the meaning of a program, but in addition, semi-eager monitoring may not be faithful. Furthermore, there are contracts which are not idempotent with this monitoring strategy.
- Lazy contract monitoring may not be faithful. Moreover, a fully correct implementation requires either a whole-program transformation or changes to the underlying runtime system. Implementing lazy contract monitoring as a library only yields an under-approximation of the monitoring strategy that discovers fewer violation than ideally detectable.

The question is now: Should we abandon contract monitoring for lazy languages at all, or is there a monitoring strategy that performs sufficiently well in practice? Our answer to this question rests on the fundamental principle that *contracts should not lie*. If a non-function contract does not signal a contract violation, then all predicates of the contract should be true, as the programmer may and should rely on this fact. Example (3) in Section 3.2 demonstrates that violating this principle may lead to subtle bugs. Even worse, unfaithful contracts force programmers to insert extra, often redundant checks to ensure that all preconditions really hold. But this contradicts the initial purpose of contracts [Mey92a]!

Hence, eager contract monitoring is preferable over semi-eager and lazy monitoring because it is the only faithful strategy presented. Its only disadvantage is that adding contracts to a program may increase the degree of strictness of a program. However, defensive programming—where the programmer checks all assertions explicitly—has exactly the same effect. These observations strongly suggest that eager contract monitoring should be used in practice.

6 Related Work

Contracts for strict functional languages. Researchers initially investigated contracts for functional languages in a strict setting. Our work, however, considers contracts for lazy functional languages.

Findler and Felleisen [FF01] give a semantic account of contract monitoring in an object-oriented language. Most of the difficulties with contracts in object-oriented languages arise from subtyping, which our language does not have.

A follow-up article [FF02] provides a contract system for higher-order functions. Their contract calculus λ^{CON} extends the λ-calculus with contracts, similar to our language $\lambda^{?!}$. In addition to the features of $\lambda^{?!}$, their calculus λ^{CON} includes first-class contracts and blame assignment. These extensions should be orthogonal to the aspects discussed in this paper. Blume and McAllester [BM04] prove soundness and completeness for a system that is equivalent to the system of Findler and Felleisen.

Findler and Blume [FB06] model contracts by pairs of domain projections. They provide an ordering relation on contracts and give further insights on the most permissive contract. In addition, they address the problem of proper blame assignment. Like our work, their model is also driven by the central properties of meaning preservation and idempotence.

Contracts for lazy functional languages. Chitil et al. [CMR04] argue that an assertion should not force extra evaluation in a lazy language like Haskell and define and implement a language of lazy assertions. Essentially, this corresponds to an approximation of what we call lazy contract monitoring for non-function contracts. Our implementation of lazy monitoring is novel and significantly different from theirs, for instance, they rely on threads to evaluate contracts. Hence, their system defers evaluation of assertions until the assertion checking thread runs, whereas our implementation guarantees that contracts are enforced as soon as all values involved are available.

Inspired by this work, Chitil and Huch [CH07b] define a pattern logic for specifying lazy assertions. Their strategy for evaluating assertions also approximates lazy contract monitoring. Chitil and Huch's pattern language can express richer assertions than our system with lazy contract monitoring. For example, they are able to express assertions on recursively defined data structures. In subsequent work [CH07a], the authors refine their interface to a monadic one and make the assertions prompt, in the sense that they signal failure as soon as a data structure is sufficiently evaluated to make an assertion fail. The style of monitoring advocated by these authors is very close to lazy monitoring. While it has the undesirable properties that we pinpoint in this work, there may be valid uses of this techniques unless the assertions used exhibit the pathological behavior of our examples.

Hinze et al. [HJL06] introduce a DSL for contracts in Haskell. Their strategy for contract monitoring roughly corresponds to the semi-eager approach discussed in this paper. Our implementations of eager and lazy contract monitoring are based on their DSL.

In recent work, Xu et al. [XPC09] present a static contract system for Haskell. Although their system is statically checked at compile time, their contracts nevertheless exhibit very similar properties to the eager monitoring style that we advocate in this paper. Hence, their specification of contracts would be suitable for dynamic (eager) monitoring, too.

Another article by the present authors [DTW09] discusses differences of contract monitoring under call-by-name and call-by-value evaluation strategies. That article studies restrictions that may be imposed on contracts to guarantee the MP and IP properties using a translation into a monadic meta-language.

7 Conclusion

We have established a taxonomy for approaches to contract monitoring in lazy languages. We have assessed each approach along the dimensions of meaning reflection, meaning preservation, faithfulness, idempotence, and implementability. The result is that there is no silver bullet for contract monitoring in lazy languages because there is no approach that fulfills all five requirements.

However, we have presented evidence that the eager monitoring approach is best in line with established theory and practice of the design by contract methodology. Although it does not preserve the semantics, it turns out to be the only approach that guarantees faithfulness. As faithfulness means that—at the end of a program run—all contract predicates are true, this approach has our strong support. Thus, contract monitoring for lazy languages should be taken to implement the slogan:

> Faithfulness is better than laziness.

References

[AK03] Parker Abercrombie and Murat Karaorman. jContractor: Design by Contract for Java. `http://jcontractor.sourceforge.net/`, 2003.

[BM04] Matthias Blume and David McAllester. A sound (and complete) model of contracts. In Kathleen Fisher, editor, *Proc. ICFP 2004*, pages 189–200, Snowbird, Utah, USA, September 2004. ACM Press, New York.

[CH07a] Olaf Chitil and Frank Huch. Monadic Prompt Lazy Assertions in Haskell. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007*, number 4807 in LNCS, pages 38–53. Springer, November 2007.

[CH07b] Olaf Chitil and Frank Huch. A Pattern Logic for Prompt Lazy Assertions. In *Implementation and Application of Functional Languages, 18th International Workshop, IFL 2006*, number 4449 in LNCS, April 2007.

[CMR04] Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy Assertions. In Phil Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, number 3145 in LNCS. Springer, November 2004.

[DTW09] Markus Degen, Peter Thiemann, and Stefan Wehr. Eager and Delayed Contract Monitoring for Call-by-value and Call-by-name Evaluation, 2009. Submitted.

[FB06] Robert Bruce Findler and Matthias Blume. Contracts as Pairs of Projections. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, April 2006. Springer.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract Soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In Simon Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

[Gil00] Andrew Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Haskell2000*, 2000.

[HJL06] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed Contracts for Functional Programming. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, April 2006. Springer.

[Kra98] Reto Kramer. iContract — the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295– 307, Los Alamitos, CA, USA, 1998.

[Lau93] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. 1993 ACM Symp. POPL*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.

[Mey92a] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[Pey92] Simon L Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *J. Funct. Program.*, 2(2):127–202, April 1992.

[PLT05] PLT Group. *PLT MzLib: Libraries Manual*. Rice University, University of Utah, December 2005. Version 300.

[XPC09] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static Contract Checking for Haskell. In Benjamin Pierce, editor, *Proc. 36th ACM Symp. POPL*, Savannah, GA, USA, January 2009. ACM Press.